Projeto *Kiosk-IUL* (Parte 2)

O presente trabalho visa aplicar os conhecimentos adquiridos durante as aulas de Sistemas Operativos e será composto por três partes, com o objetivo de desenvolver os diferentes aspetos da plataforma **Kiosk-IUL**. Iremos procurar minimizar as interdependências entre partes do trabalho.

Este enunciado detalha apenas as funcionalidades que devem ser implementadas na parte 2 do trabalho.

A aplicação **Kiosk-IUL** permite gerir e fazer compras num quiosque de mercearias no campus do ISCTE que está aberto 24h e é totalmente automatizado. Na aplicação **Kiosk-IUL** existem os seguintes conceitos:

Algumas definições básicas e tipos de dados para interoperabilidade entre **Cliente** e **Servidor** estão no ficheiro **common.h**:

```
#define MIN_PROCESSAMENTO 1      // Tempo mínimo de processamento do Cliente
#define MAX_PROCESSAMENTO 7      // Tempo máximo de processamento do Cliente
#define MAX_ESPERA        5      // Tempo máximo de espera por parte do Cliente

typedef struct {
    int nif;                     // Número de contribuinte do utilizador
    char senha[20];              // Senha do utilizador
    char nome[52];               // Nome do utilizador
    int saldo;                   // Saldo do utilizador
    int pid_cliente;             // PID do processo Cliente
    int pid_servidor_dedicado;   // PID do processo Servidor Dedicado
} Login;
```

Os alunos deverão, em vez de **printf** (não será analisado para efeito de avaliação), utilizar sempre as macros **`so_success`** (para as mensagens de sucesso) e **`so_error`** (para as mensagens de erro) definidas no *header file* **/home/so/reference/so_utils.h** (cuja sintaxe está descrita na KB C Language & Compiling / Para que serve o header file "/home/so/reference/so_utils.h?"), indicando SEMPRE a alínea correspondente (e.g., **`so_error("S2");`**), e sempre que no enunciado estiverem indicados os pedidos de valores entre **< >**, o aluno deverá (naturalmente!) substituir esse texto pelos valores indicados (e.g., **`so_success("S4", "%d", pid_servidor);`**), garantindo que é sempre cumprida estritamente a especificação apresentada sem acrescentar mais informação.

A *baseline* para o trabalho encontra-se no Tigre, na diretoria **`/home/so/trabalho-2022-2023/parte-2`**. É obrigatório que os alunos trabalhem com base **nestes** ficheiros e não com base em ficheiros vazios.

- Para tal, **EXECUTE**, **a partir da sua diretoria local de projeto**, os seguintes comandos:

```
$ cp -r /home/so/trabalho-2022-2023/parte-2 .
$ cp -r /home/so/trabalho-2022-2023/parte-2/utils .
```

Resultado:

```
bd_utilizadores.dat
cliente.c
common.h -> utils/common.h
Makefile
servidor.c
so-2022-trab2-validator/
so_utils.h -> /home/so/reference/so_utils.h
utils -> /home/so/trabalho-2022-2023/utils/parte-2/
```

Procedimento de entrega e submissão do trabalho

O trabalho de SO será realizado **individualmente**, logo sem recurso a grupos.

A entrega da Parte 2 do trabalho será realizada através da criação de **um** ficheiro ZIP cujo nome é o nº do aluno, e.g., "**a<nºaluno>-parte-2.zip**" (**ATENÇÃO:** não serão aceites ficheiros RAR, 7Z ou outro formato) onde estarão todos os ficheiros criados. Estes serão **apenas** os ficheiros de código, ou seja, na parte 2, apenas os ficheiros (***.c *.h**).

Cada um dos módulos será desenvolvido com base nos ficheiros fornecidos, e que estão na diretoria do Tigre **"/home/so/trabalho-2022-2023/parte-2"**, e deverá incluir nos comentários iniciais um "relatório" indicando a descrição do módulo e explicação do mesmo (poderá ser muito reduzida se o código tiver comentários bem descritivos).

Para criarem o ficheiro ZIP para submissão do trabalho, posicionem-se no Tigre na diretoria **parte-2**, e executem:

```
$ zip $USER-parte-2.zip *.c *.h
```

O ficheiro ZIP deverá depois ser transferido do Tigre para a vossa área local (Windows/Linux/Mac) via SFTP, para depois ser submetido via Moodle (ver no Moodle a KB Basics / Criar ficheiro ZIP para submeter trabalho no Moodle).

Antes de submeter, por favor validem que o ficheiro ZIP não inclui diretorias ou ficheiros extra indesejados.

A entrega desta parte do trabalho deverá ser feita por via eletrónica, através do Moodle:

- Moodle da UC Sistemas Operativos, Selecionam a opção sub-menu "Quizzes & Assignments";
- Selecionem o link "Submit SO Assignment 2022-2023 Part 2";
- Dentro do formulário, selecionem o botão "Enviar trabalho" e anexem o vosso ficheiro .zip (a forma mais fácil é simplesmente fazer via *drag-and-drop*) e selecionar o botão "Guardar alterações". Podem depois mais tarde ressubmeter o vosso trabalho as vezes que desejarem, enquanto estiverem dentro do prazo para entrega do trabalho. Para isso, na mesma opção, pressionar o botão "Editar submissão", selecionar o ficheiro, e depois o botão "Apagar", sendo que depois pode arrastar o novo ficheiro e pressionar "Guardar alterações". **Apenas a última submissão será contabilizada**. Certifiquem-se que a submissão foi concluída, e que esta última versão tem todas as alterações que desejam entregar dado que os docentes apenas considerarão esta última submissão;
- Avisamos que a hora *deadline* acontece sempre poucos **minutos antes da meia-noite**, pelo que se urge a que os alunos não esperem por essa hora final para entregar e o façam antes, idealmente um dia antes, ou no pior dos casos, pelo menos uma hora antes. **Não serão consideradas válidas as entregas realizadas por e-mail**. Poderão testar a entrega nos dias anteriores para perceber se há algum problema com a entrega, sendo que, **apenas a última submissão conta**.

Política em caso de fraude

O trabalho corresponde ao esforço individual de cada aluno. São consideradas fraudes as seguintes situações: Trabalho parcialmente copiado, facilitar a cópia através da partilha de ficheiros, ou utilizar material alheio sem referir a fonte.

Em caso de deteção de fraude, os trabalhos em questão não serão avaliados, sendo enviados à Comissão Pedagógica da escola (ISTA) ou ao Conselho Pedagógico do ISCTE, consoante a gravidade da situação, que decidirão a sanção a aplicar aos alunos envolvidos. Serão utilizadas as ferramentas *Moss* e *SafeAssign* para deteção automática de cópias.

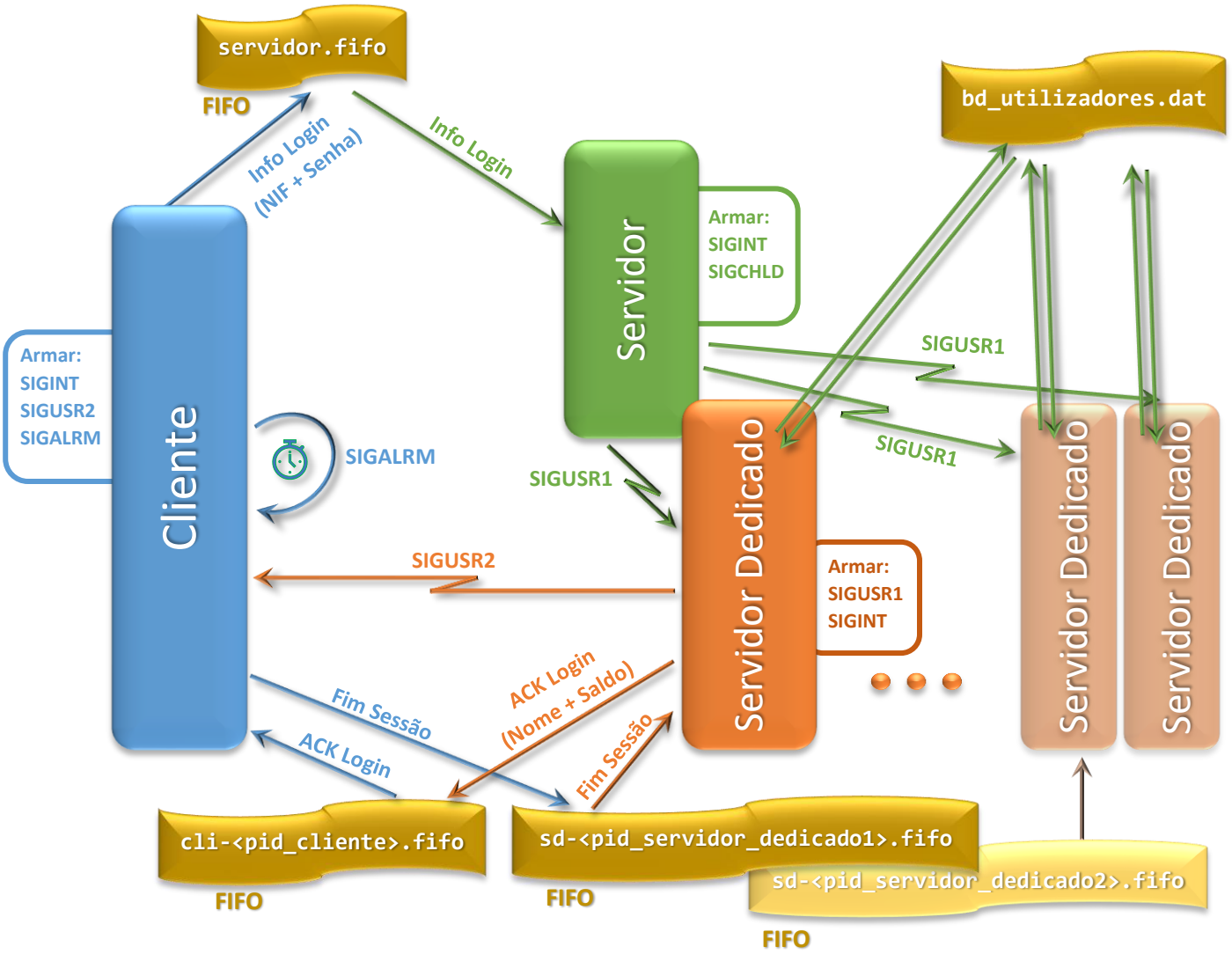
Recorda-se ainda que o Anexo I do Código de Conduta Académica, publicado a 25 de janeiro de 2016 em Diário da República, 2ª Série, nº 16, indica no seu ponto 2 que quando um trabalho ou outro elemento de avaliação apresentar um nível de coincidência elevado com outros trabalhos (percentagem de coincidência com outras fontes reportada no relatório que o referido software produz), cabe ao docente da UC, orientador ou a qualquer elemento do júri, após a análise qualitativa desse relatório, e em caso de se confirmar a suspeita de plágio, desencadear o respetivo procedimento disciplinar, de acordo com o Regulamento Disciplinar de Discentes do ISCTE - Instituto Universitário de Lisboa, aprovado pela deliberação nº 2246/2010, de 6 de dezembro.

O ponto 2.1 desse mesmo anexo indica ainda que no âmbito do Regulamento Disciplinar de Discentes do ISCTE-IUL, são definidas as sanções disciplinares aplicáveis e os seus efeitos, podendo estas variar entre a advertência e a interdição da frequência de atividades escolares no ISCTE-IUL até cinco anos.

Parte 2 – Processos e Sinais

Data de entrega: 25 de abril de 2023

Nesta parte do trabalho, será implementado um modelo simplificado de gestão da autenticação de um quiosque de compras automáticas, **Kiosk-IUL**, baseado em comunicação por sinais entre processos, utilizando a linguagem de programação C. Considere o seguinte diagrama, que apresenta uma visão geral da arquitetura pretendida:



Pretende-se, nesta fase, simular um servidor de sessões de autenticação de utilizadores no quiosque **Kiosk-IUL**. Assim, teremos dois módulos – **Cliente** e **Servidor**. Os conhecimentos que se pretende que os alunos sejam avaliados com este trabalho são:

- Criação de processos e concorrência entre processos (**fork** e **wait**)
- Interação entre processos usando sinais (**kill**, **signal** e **alarm**);
- Manuseamento de ficheiros de acesso sequencial (**so_fgets**, **fprintf**) e direto (**fread**, **fwrite**, **fseek**);
- Interação com o utilizador (**printf**, **so_gets**, **so_geti**)
- Comunicação usando Named Pipes ou FIFOs (**S_ISFIFO**, **stat**).

Atenção: Apesar do vários ficheiros necessários para a realização do trabalho serem fornecidos na diretoria do Tigre "**/home/so/trabalho-2022-2023/parte-2**", assume-se que, para a sua execução, os scripts e todos os ficheiros de input e de output estarão todos **sempre** presentes na mesma diretoria, que não deve estar *hard-coded*, ou seja, os programas entregues devem correr em qualquer diretoria.

1. Módulo: `servidor.c`

O módulo **Servidor** é responsável pelo processamento das autenticações dos utilizadores. Está dividido em duas partes, o **Servidor** (pai) e zero ou mais **Servidores Dedicados** (filhos). Este módulo realiza as seguintes tarefas:

S1 Valida se o ficheiro **bd_utilizadores.dat** existe na diretoria local, dá **`so_error`** (i.e., `so_error("S1", "")`) e termina o **Servidor** se o ficheiro não existir. Caso contrário, dá **`so_success`** (i.e., `so_success("S1", "")`);

S2 Cria o ficheiro com organização FIFO (*named pipe*) do **Servidor**, de nome **servidor.fifo**, na diretoria local. Se houver erro na operação, dá **`so_error`** e termina o **Servidor**. Caso contrário, dá **`so_success`**;

S3 Arma e trata os sinais **SIGINT** (ver **S7**) e **SIGCHLD** (ver **S8**). Em caso de qualquer erro a armar os sinais, dá **`so_error`** e segue para o passo **S6**. Caso contrário, dá **`so_success`**;

S4 Abre o FIFO do **Servidor** para leitura, lê um pedido (acesso direto) que deverá ser um elemento do tipo **Login**, e fecha o mesmo FIFO. Se houver erro, dá **`so_error`** e reinicia o processo neste mesmo passo **S4**, lendo um novo pedido. Caso contrário, dá **`so_success <nif> <senha> <pid_cliente>`**;

S5 Cria um processo filho (**fork**) **Servidor Dedicado**. Se houver erro, dá **`so_error`**. Caso contrário, o processo **Servidor Dedicado** (filho) continua no passo **SD9**, enquanto o processo **Servidor** (pai) dá **`so_success "Servidor Dedicado: PID <pid_servidor_dedicado>"`**, e recomeça no passo **S4**;

S6 Remove o FIFO do **Servidor**, de nome **servidor.fifo**, da diretoria local. Em caso de erro, dá **`so_error`**, caso contrário, dá **`so_success`**. Em ambos os casos, termina o processo **Servidor**.

S7 O sinal armado **SIGINT** serve para o dono da loja encerrar o **Servidor**, usando o atalho **<CTRL+C>**. Se receber esse sinal (do utilizador via Shell), o **Servidor** dá **`so_success "Shutdown Servidor"`**, e faz as ações:

- **S7.1** Abre o ficheiro **bd_utilizadores.dat** para leitura. Em caso de erro na abertura do ficheiro, dá **`so_error`** e segue para o passo **S6**. Caso contrário, dá **`so_success`**;
- **S7.2** Lê (acesso direto) um elemento do tipo **Login** deste ficheiro. Em caso de erro na leitura do ficheiro, dá **`so_error`** e segue para o passo **S6**;
- **S7.3** Se o elemento **Login** lido tiver **pid_servidor_dedicado** > 0, então envia ao PID desse **Servidor Dedicado** o sinal **SIGUSR1**;
- **S7.4** Se tiver chegado ao fim do ficheiro **bd_utilizadores.dat**, fecha o ficheiro e dá **`so_success`**. Caso contrário, volta ao passo **S7.2**;
- **S7.5** Vai para o passo **S6**.

S8 O sinal armado **SIGCHLD** serve para que o **Servidor** seja alertado quando um dos seus filhos **Servidor Dedicado** terminar. Se o **Servidor** receber esse sinal, identifica o PID do **Servidor Dedicado** que terminou (usando **wait**), e dá **`so_success "Terminou Servidor Dedicado <pid_servidor_dedicado>"`**, retornando ao passo **S4** sem reportar erro;

SD9 O novo processo **Servidor Dedicado** (filho) arma os sinais **SIGUSR1** (ver **SD18**) e **SIGINT** (programa-o para ignorar este sinal). Em caso de erro a armar os sinais, dá **`so_error`** e termina o **Servidor Dedicado**. Caso contrário, dá **`so_success`**;

SD10 O **Servidor Dedicado** deve validar, em primeiro lugar, no pedido **Login** recebido do **Cliente** (herdado do processo **Servidor** pai), se o campo **pid_cliente** > 0. Se for, dá **`so_success`**, caso contrário dá **`so_error`** e termina o **Servidor Dedicado**;

SD11 Abre o ficheiro **bd_utilizadores.dat** para leitura. Em caso de erro na abertura do ficheiro, dá **`so_error`** e termina o **Servidor Dedicado**. Caso contrário, dá **`so_success`**, e faz as seguintes operações:

- **SD11.1** Inicia uma variável **index_client** com o índice (inteiro) do elemento **Login** corrente lido do ficheiro. Para simplificar, pode considerar que este ficheiro nunca terá nem mais nem menos elementos;
- **SD11.2** Se já chegou ao final do ficheiro **bd_utilizadores.dat** sem encontrar o cliente, coloca **index_client**=-1,

dá **so_error**, fecha o ficheiro, e segue para o passo SD12;

SD11.3 Caso contrário, lê (acesso direto) um elemento Login do ficheiro e incrementa a variável **index_client**. Em caso de erro na leitura do ficheiro, dá **so_error** e termina o Servidor Dedicado;

SD11.4 Valida se o **NIF** passado no pedido do Cliente corresponde ao **NIF** do elemento Login do ficheiro. Se não corresponder, então reinicia ao passo SD11.2;

SD11.5 Se, pelo contrário, os **NIF**s correspondem, valida se a **Senha** passada no pedido do Cliente bate certo com a **Senha** desse mesmo elemento Login do ficheiro. Caso isso seja verdade, então dá **so_success <index_client>**. Caso contrário, dá **so_error** e coloca **index_client**=-1;

SD11.6 Termina a pesquisa, e fecha o ficheiro **bd_utilizadores.dat**.

SD12 Modifica a estrutura Login recebida no pedido do Cliente: se **index_client** < 0, então preenche o campo **pid_servidor_dedicado**=-1, e segue para o passo SD13. Caso contrário (**index_client** >= 0):

SD12.1 Preenche os campos **nome** e **saldo** da estrutura Login recebida no pedido do Cliente com os valores do item de **bd_utilizadores.dat** para **index_client**. Preenche o campo **pid_servidor_dedicado** com o PID do processo Servidor Dedicado, ficando assim a estrutura Login completamente preenchida;

SD12.2 Abre o ficheiro **bd_utilizadores.dat** para escrita. Em caso de erro na abertura do ficheiro, dá **so_error** e termina o Servidor Dedicado. Caso contrário, dá **so_success**;

SD12.3 Posiciona o apontador do ficheiro (**fseek**) para o elemento Login correspondente a **index_client**, mais precisamente, para imediatamente antes dos campos a atualizar (**pid_cliente** e **pid_servidor_dedicado**). Em caso de erro, dá **so_error** e termina. Caso contrário, dá **so_success**;

SD12.4 Escreve no ficheiro (acesso direto), na posição atual, os campos **pid_cliente** e **pid_servidor_dedicado** atualizando assim a estrutura Login correspondente a este Cliente na base de dados, e fecha o ficheiro. Em caso de erro, dá **so_error** e termina. Caso contrário, dá **so_success**.

SD13 Cria o ficheiro com organização FIFO (*named pipe*) **sd-<pid_servidor_dedicado>.fifo** na diretoria local. Se houver erro na operação, dá **so_error**, e termina o Servidor Dedicado. Caso contrário, dá **so_success**;

SD14 Abre o FIFO do Cliente, de nome **cli-<pid_cliente>.fifo** na diretoria local, para escrita, escreve (acesso direto) no FIFO do Cliente a estrutura Login recebida no pedido do Cliente, e fecha o mesmo FIFO. Em caso de erro, dá **so_error**, e segue para o passo SD17. Caso contrário, dá **so_success**.

SD15 Abre o FIFO do Servidor Dedicado, lê uma string enviada pelo Cliente, e fecha o mesmo FIFO. Em caso de erro, dá **so_error**, e segue para o passo SD17. Caso contrário, dá **so_success <string enviada>**.

SD16 Modifica a estrutura Login recebida no pedido do Cliente, por forma a terminar a sessão:

SD16.1 Preenche os campos **pid_cliente**=-1 e **pid_servidor_dedicado**=-1;

SD16.2 Abre o ficheiro **bd_utilizadores.dat** na diretoria local para escrita. Em caso de erro na abertura do ficheiro, dá **so_error**, e segue para o passo SD17. Caso contrário, dá **so_success**;

SD16.3 Posiciona o apontador do ficheiro (**fseek**) para o elemento Login correspondente a **index_client**, mais precisamente, para imediatamente antes dos campos **pid_cliente** e **pid_servidor_dedicado**. Em caso de erro, dá **so_error**, e segue para o passo SD17. Caso contrário, dá **so_success**;

SD16.4 Escreve no ficheiro (acesso direto), na posição atual, os campos **pid_cliente** e **pid_servidor_dedicado** atualizando assim a estrutura Login deste Cliente na base de dados, e fecha o ficheiro. Em caso de erro, dá **so_error**, caso contrário, dá **so_success**. Em ambos casos, segue para o passo SD17;

SD17 Remove o FIFO **sd-<pid_servidor_dedicado>.fifo** da diretoria local. Em caso de erro, dá **so_error**, caso contrário, dá **so_success**. Em ambos os casos, termina o processo Servidor Dedicado.

SD18 O sinal armado SIGUSR1 serve para que o Servidor Dedicado seja alertado quando o Servidor principal quer terminar. Se o Servidor Dedicado receber esse sinal, envia um sinal SIGUSR2 ao Cliente (para avisá-lo do Shutdown), dá **so_success**, e vai para o passo SD16 para terminar *clean* o Servidor Dedicado.

2. Módulo: `cliente.c`

O módulo **Cliente** é responsável pela interação com o utilizador. Após o login do utilizador, este poderá realizar atividades durante o tempo da sessão. Assim, definem-se as seguintes tarefas a desenvolver:

C1 Valida se o ficheiro com organização FIFO (*named pipe*) do **Servidor**, de nome **servidor.fifo,** existe na diretoria local. Se esse FIFO não existir, dá **`so_error`** e termina o **Cliente**. Caso contrário, dá **`so_success`**;

C2 Arma e trata os sinais **SIGUSR2** (ver **C12**), **SIGINT** (ver **C13**), e **SIGALRM** (ver **C14**). Em caso de qualquer erro a armar os sinais, dá **`so_error`** e termina o **Cliente**. Caso contrário, dá **`so_success`**;

C3 Pede ao utilizador que preencha os dados referentes à sua autenticação (**NIF** e **Senha**), criando um elemento do tipo **Login** com essas informações, e preenchendo também o campo **pid_cliente** com o PID do seu próprio processo **Cliente**. Os restantes campos da estrutura **Login** não precisam ser preenchidos. Em caso de qualquer erro, dá **`so_error`** e termina o **Cliente**. Caso contrário dá **`so_success <nif> <senha> <pid_cliente>`**;

C4 Cria o ficheiro com organização FIFO (*named pipe*) do **Cliente**, de nome **cli-<pid_cliente>.fifo**, na diretoria local. Se houver erro na operação, dá **`so_error`** e termina o **Cliente**. Caso contrário, dá **`so_success`**;

C5 Abre o FIFO do **Servidor** (**servidor.fifo**), escreve as informações do elemento **Login** (acesso direto) nesse FIFO do **Servidor**, e fecha o mesmo FIFO. Em caso de erro na escrita, dá **`so_error`**, e vai para o passo **C11**, caso contrário, dá **`so_success`**;

C6 Configura um alarme com o valor de MAX_ESPERA segundos (ver **C14**), e dá **`so_success "Espera resposta em <MAX_ESPERA> segundos"`**;

C7 Abre o FIFO do **Cliente** para leitura, lê a informação do FIFO **Cliente** (acesso direto), que deverá ser um elemento do tipo **Login**, e fecha o mesmo FIFO. Se houver erro na operação, dá **`so_error`**, e vai para o passo **C11**. Caso contrário, dá **`so_success <nome> <saldo> <pid_servidor_dedicado>`**;

C8 "Desliga" o alarme configurado em **C6**. Valida se o resultado da autenticação do **Servidor Dedicado** foi sucesso (convenciona-se que se a autenticação não tiver sucesso, o campo **pid_servidor_dedicado**==-1). Se a autenticação não foi bem-sucedida, dá **`so_error`**, e vai para o passo **C11**. Caso contrário, dá **`so_success`**;

C9 Calcula um valor aleatório (usando **so_rand()**) de tempo entre os valores MIN_PROCESSAMENTO e MAX_PROCESSAMENTO, dá **`so_success "Processamento durante <Tempo> segundos"`**, e aguarda esse valor em segundos (**sleep**);

C10 Abre o FIFO do **Servidor Dedicado**, de nome **sd-<pid_servidor_dedicado>.fifo** para escrita na diretoria local, escreve nesse FIFO (acesso sequencial) a string "Sessão Login ativa durante <Tempo> segundos", e fecha o mesmo FIFO. Em caso de erro, dá **`so_error`**. Caso contrário, dá **`so_success`**. Em ambos os casos, vai para o passo **C11**;

C11 Remove o FIFO do **Cliente**, de nome **cli-<pid_cliente>.fifo**, da diretoria local. Em caso de erro, dá **`so_error`**, caso contrário, dá **`so_success`**. Em ambos os casos, termina o processo **Cliente**.

C12 O sinal armado **SIGUSR2** serve para o **Servidor Dedicado** indicar que o servidor está em modo *shutdown*. Se o **Cliente** receber esse sinal, dá **`so_success`** e prossegue para o passo **C11**.

C13 O sinal armado **SIGINT** serve para que o utilizador possa cancelar o pedido do lado do **Cliente**, usando o atalho **<CTRL+C>**. Se receber esse sinal (do utilizador via Shell), o **Cliente** dá **`so_success "Shutdown Cliente"`**, e depois, abre o FIFO do **Servidor Dedicado**, escreve nesse FIFO (acesso sequencial) a string "Sessão cancelada pelo utilizador", e fecha o mesmo FIFO. Em caso de erro, dá **`so_error`**. Caso contrário, dá **`so_success`**. Em ambos os casos, vai para o passo **C11**;

C14 O sinal armado **SIGALRM** serve para que, se o **Cliente** em **C7** esperou mais do que MAX_ESPERA segundos sem resposta, o **Cliente** dá **`so_error "Timeout Cliente"`**, e prossegue para o passo **C11**.

Anexo A: Scripts de suporte ao trabalho

Scripts **fornecidos,** com Mensagens de **sucesso**, **erro, debug** e **validação de programas**:

Mensagens de output com Erro (com exemplos): Macro `so_error(<passo>, <Mensagem>, [...])`

A sintaxe dos argumentos "`<Mensagem>, [...]`" é semelhante à de `printf()`; esta macro, tem como output no STDOUT:
"`@ERROR {<Passo>} [<Mensagem>, [...]]`"

Exemplos de invocação:

- Em **S1**, O ficheiro **bd_utilizadores.dat** não existe:
  - `so_error("S1", "");`
- Em **C14**, O **Cliente** deu Timeout:
  - `so_error("C13", "Timeout Cliente");`

Mensagens de output com Sucesso (com exemplos): Macro `so_success(<passo>, <Mensagem>, [...])`

A sintaxe dos argumentos "`<Mensagem>, [...]`" é semelhante à de `printf()`; esta macro, tem como output no STDOUT:
"`@SUCCESS {<Passo>} [<Mensagem>, [...]]`"

Exemplos de invocação:

- Em **S3**, o **Servidor** armou corretamente os sinais SIGINT e SIGCHLD:
  - `so_success("S3", "");`
- Em **C6**, o **Cliente** indica que iniciou o período de espera:
  - `so_success("C6", "Espera resposta em %d segundos", MAX_ESPERA);`
- Em **SD11.5**, o **Servidor Dedicado** indica que descobriu a entrada do cliente no ficheiro **bd_utilizadores.dat**:
  - `so_success("SD11.3", "%d", index_client);`
- Em **S4**, o **Servidor** leu um pedido do **Cliente** na forma de um elemento **Login**:
  - `so_success("S4", "%d %s %d", request.nif, request.senha, request.pid_cliente);`

Mensagens de Debug: Apesar de não ser necessário, disponibilizou-se também uma macro para as mensagens de debug dos scripts, dado que será muito útil aos alunos: **Macro `so_debug(<Mensagem>, [...])`**

A sintaxe dos argumentos "`<Mensagem>, [...]`" é semelhante à de `printf()`; esta macro, tem como output no STDOUT:
"`@DEBUG:<Source file>:<line>:<function>: [<Mensagem>, [...]]`"

Exemplos de invocação:

- Em **SD11.3**, para ver os valores da entrada atual lida da BD:
  - `so_debug("Entrada atual BD: NIF: %d; Senha: %s", itemDB.nif, itemDB.senha);`
- Em **C7**, simplesmente para indicar um teste de passagem por uma parte do código:
  - `so_debug("Passei por aqui");`

Tem a vantagem de que mostra sempre as mensagens de debug (não precisa sequer ser nunca apagado). Quando os alunos quiserem apagar as mensagens de debug, basta descomentar a seguinte linha do programa atual:

```
//#define SO_HIDE_DEBUG       // Uncomment this line to hide all @DEBUG statements
```

E, assim, não precisam de apagar as invocações à macro so_debug, mantendo os vossos programas intocados.

Manipulação de ficheiros binários:

Neste trabalho são armazenadas informações num ficheiro em formato binário, **bd_utilizadores.dat**. Não é fácil visualizar este ficheiro usando a aplicação **cat**. Uma das formas sugeridas de analisar estes ficheiros é usando as aplicações **hexdump** ou **xxd**. No entanto, para facilitar esta tarefa, foi fornecido um script que ajuda a visualizar os conteúdos deste ficheiro (ver KB Moodle correspondente), que estão de acordo com a estrutura **Login**:

```
typedef struct {
    int nif;                    // Número de contribuinte do utilizador
    char senha[20];             // Senha do utilizador
    char nome[50];              // Nome do utilizador
    int saldo;                  // Saldo do utilizador
    int pid_cliente;            // PID do processo Cliente
    int pid_servidor_dedicado;  // PID do processo Servidor Dedicado
} Login;
```

Assim, o comando:

```
$ ./utils/so_show-binary-login.sh bd_utilizadores.dat
```

Mostra o resultado:

```
> ./utils/so_show-binary-login.sh bd_utilizadores.dat
| 235123532 | qwerty | Paulo Pereira | 123 | -1 | -1 |
| 234580880 | 12qwaszx | Catarina Cruz | 50 | -1 | -1 |
| 215654377 | 09polkmn | Joao Baptista Goncalves | 20 | -1 | -1 |
```

Da mesma forma, foi desenvolvida uma ferramenta utilitária que lê informações de um ficheiro de texto, sendo que nesse ficheiro de texto, cada linha de texto corresponde a um registo, em que cada um dos campos desse registo são separados por um caracter separador, e produz como output um ficheiro binário de elementos do formato **Login** acima descrita.

Assim, o comando:

```
$ ./utils/so_generate-binary-login.exe utilizadores.txt : bd_utilizadores.dat
```

Lê os utilizadores que estão no ficheiro utilizadores.txt:

```
235123532:qwerty:Paulo Pereira:123:-1:-1
234580880:12qwaszx:Catarina Cruz:50:-1:-1
215654377:09polkmn:Joao Baptista Goncalves:20:-1:-1
```

sendo que cada registo de utilizador, como se pode ver acima, ocupa uma linha de texto, e os vários campos do registo estão separados pelo caracter ':', e produz como output o ficheiro **bd_utilizadores.dat** (que é o mesmo ficheiro que foi fornecido aos alunos). O ficheiro **utilizadores.txt** pode ser encontrado na diretoria "utils".

Poderá também editar este ficheiro de texto para produzir outro de formato similar, que tenha apenas um utilizador, e assim produzir um ficheiro binário com apenas um elemento **Login**, por exemplo, o ficheiro **pedido-cliente.txt**:

```
234580880:12qwaszx:sem_nome:0:123456:-1
```

Este ficheiro de texto define um utilizador com os dados (NIF e Senha) de um utilizador existente na BD (Catarina Cruz), e com um pid_cliente de valor fictício de 123456. Se agora usarmos esta ferramenta desenvolvida:

```
$ ./utils/so_generate-binary-login.exe pedido-cliente.txt : pedido-cliente1.dat
```

Ficamos com um ficheiro **pedido-cliente1.dat**. Atenção: este procedimento apenas mostra como o processo pode ser realizado, o ficheiro pedido-cliente.txt e o ficheiro pedido-cliente1.dat não são fornecidos aos alunos, se estes quiserem, podem facilmente criá-los da maneira que está descrita neste procedimento. Este ficheiro pode ser útil, por exemplo se o aluno estiver a fazer o programa **Servidor** e não quiser estar a fazer ao mesmo tempo o **Cliente**, pode simular o comportamento do módulo **Cliente** usando a linha de comandos, desta forma:

- Lançamos o processo **Servidor**, cujo comportamento esperado (como indicado no enunciado) é criar o FIFO **servidor.fifo**, e a seguir, ler desse FIFO um elemento binário do tipo **Login**. O processo **Servidor** irá então bloquear até que alguém escreva nesse FIFO o elemento;
- Em vez de desenvolvermos o programa **Cliente**, podemos simplesmente simular o seu comportamento usando a linha de comandos. O comportamento do **Cliente** é, de acordo com o enunciado:
    - Criar um FIFO **Cliente**, de nome "**cli-<PID_Cliente>.fifo**". Para a nossa simulação, vamos criar um FIFO com o PID_Cliente fictício acima indicado 123456, usando o comando **`mkfifo cli-123456.fifo`**;
    - Agora, temos de "escrever" no FIFO do **Servidor** o elemento **Login** que criámos acima de forma fictícia. Para tal, assumindo que o FIFO **servidor.fifo** já foi criado anteriormente pelo processo **Servidor**, basta simplesmente executarmos o comando **`cat pedido-cliente1.dat > servidor.fifo`**;
    - Assumindo que o **Servidor** tem o comportamento especificado no enunciado, irá criar um processo **Servidor Dedicado**, que vai escrever no nosso FIFO **Cliente** fictício um elemento **Login** já com os dados atualizados do **Servidor Dedicado** (Nome: Catarina Cruz, Saldo: 50, e pid_servidor_dedicado). Como podemos ler essa informação? Executando o comando **`cat cli-123456.fifo > ackLogin.dat`**, e depois podemos ver a informação escrita neste ficheiro temporário **ackLogin.dat**, usando o comando **`./utils/so_show-binary-login.sh ackLogin.dat`**;
    - Daqui podemos tirar o valor do PID do **Servidor Dedicado**, que servirá para de seguida podermos escrever no seu FIFO, quem de acordo com a especificação indicada no enunciado, terá o nome **sd-<pid_servidor_dedicado>.fifo** uma mensagem de finalização "Sessão Login ativa durante <Tempo> segundos" ou então "Sessão cancelada pelo utilizador" (conforme o teste pretendido), executando: **`echo "Sessão Login ativa durante 30 segundos" > sd-<pid_servidor_dedicado>.fifo`** (já que esta informação é suposto ir em formato de texto e não binária).

De forma análoga, podemos testar o programa **Cliente**, igualmente simulando o comportamento do módulo **Servidor** usando a linha de comandos, desta forma:

- Em vez de desenvolvermos o programa **Servidor** ao mesmo tempo que o Cliente, podemos simular o seu comportamento usando a linha de comandos. O comportamento do **Servidor** é, de acordo com o enunciado:
    - Criar um FIFO **Servidor**, de nome "**pedidos.fifo**". Para a nossa simulação, basta executar o comando na linha de comandos **`mkfifo pedidos.fifo`**;
    - Assumindo que o **Cliente** tem o comportamento especificado no enunciado, irá pedir os dados de NIF e Senha ao utilizador, escrevendo no nosso FIFO **Servidor** fictício um elemento **Login** com esses dados. Como podemos ler essa informação? Executando **`cat servidor.fifo > pedido.dat`**, e depois podemos ver a informação escrita neste ficheiro temporário **pedido.dat**, usando o comando **`./utils/so_show-binary-login.sh pedido.dat`**;
    - Daqui podemos tirar o valor do PID do **Cliente**, que servirá para de seguida podermos escrever no seu FIFO, quem de acordo com a especificação indicada no enunciado, terá o nome **cli-<pid_cliente>.fifo** um elemento **Login** que criaremos de forma fictícia, criando um ficheiro de texto **ackLogin.txt**, que terá as informações do **Cliente**, e inserindo um valor fictício 234567 para pid_servidor_dedicado, e

depois correr **`./utils/so_generate-binary-login.exe ackLogin.txt : ackLogin.dat`**. Com isto, ficamos com o nosso elemento **Login** de resposta de **Servidor Dedicado** simulado;

- De seguida, o nosso **Servidor Dedicado** simulado deve criar um FIFO usando o pid_servidor_dedicado, de valor fictício 234567, usando o comando **`mkfifo sd-234567.fifo`**;
- Agora, temos de "escrever" no FIFO do **Cliente** o elemento **Login** que criámos acima de forma fictícia. Para tal, assumindo que o FIFO **cli-<pid_cliente>.fifo** já foi criado anteriormente pelo processo **Cliente**, basta simplesmente executarmos o comando **`cat ackLogin.dat > cli-<pid_cliente>.fifo`**;
- Finalmente, resta-nos ler a mensagem de finalização enviada pelo **Cliente** usando simplesmente o comando **`cat sd-234567.fifo`** (já que esta informação é suposta ir em formato de texto e não binária).

Para os restantes testes, poderão usar as ferramentas produzidas para criar elementos Login com valores diferentes, para assim produzir resultados diferentes. Esta forma destina-se apenas a ajudar a testar os vossos programas Cliente e Servidor, e de forma alguma se destinam a substituir o projeto em linguagem C.

Script Validador do trabalho:

Como anunciado nas aulas, está disponível para os alunos um script de validação dos trabalhos, para os alunos terem uma noção dos critérios de avaliação utilizados.

Passos para realizar a validação do vosso trabalho:

- Garantam que o vosso trabalho (i.e., os ficheiros `*.c *.h`) está localizado numa diretoria local da vossa área. Para os efeitos de exemplo para esta demonstração, assumiremos que essa diretoria terá o nome **`parte-2`** (mas poderá ser outra qualquer);
- Posicionem-se nessa diretoria **`parte-2`** da vossa área:

  ```
  $ cd parte-2
  ```

- Deem o comando $ **`pwd`** , e validem que estão mesmo na diretoria correta;
- Deem o comando $ **`ls -l`** , e confirmem que todos os ficheiros **`*.c *.h`** do vosso trabalho estão mesmo nessa diretoria, e também está a diretoria do validador **`so-2022-trab2-validator`**;
- Agora, posicionem-se na subdiretoria do validador:

  ```
  $ cd so-2022-trab2-validator/
  ```

- E, finalmente, dentro dessa diretoria, executem o script de validação do vosso trabalho, que está na diretoria "pai" (**`..`**)

  ```
  $ ./so-2022-trab2-validator.py ..
  ```

- Resta agora verificarem quais dos vossos testes "passam" (✓) e quais "chumbam" (✗);
- Façam as alterações para correção dos vossos scripts;
- Sempre que quiserem voltar a fazer nova validação, basta novamente posicionarem-se na subdiretoria **`so-2022-trab2-validator`** e correrem o script de validação como demonstrado acima;
- A aplicação **`so-2022-trab2-validator.py`** tem algumas opções que podem ser úteis aos alunos:
  - Se fizerem **`so-2022-trab2-validator.py -h`**, (ou **`--help`**), podem ver as várias opções;
  - Se usarem a opção **`-d`** (ou **`--debug`**), podem visualizar as mensagens que colocaram no código usando **`so_debug`**, **`so_success`**, e **`so_error`**. Caso contrário essas mensagens serão omitidas;
  - Se usarem a opção **`-e`** (ou **`--stoponerror`**), o validador irá parar assim que encontrar o primeiro erro, o que pode ser prático para não terem um output muito extenso;
  - Se usarem a opção **`-s`** (ou **`--server`**), o validador apenas irá validar o servidor (ou seja, não irá validar o cliente); se usarem a opção **`-c`** (ou **`--client`**), o validador apenas irá validar o cliente (ou seja, não irá validar o servidor); se usarem as duas, nada será validado.